Gesture Detection and Reinforcement Learning For Game Bomb In Hand

Technical Paper for CSCI 527 Final

Zikai Cheng(zikaiche@usc.edu) Jingyi Wang(jwang750@usc.edu) Zunqi(Wilson) Huang(whuang06@usc.edu)

Fanzhe Meng(fanzheme@usc.edu) Yijing Xiao(yijingxi@usc.edu) Jianchen Gao(gaojianc@usc.edu)

Xuetao Sun(xuetaosu@usc.edu)

***Abstract*—This is the final technical paper for the course CSCI 527 Team Bomb In Hand. Our team designed a bombing game in which a player can control her/his character by making gestures in real world. The player is able to hold or throw bombs and fight against an opponent agent. We achieved that by creating a gesture recognition pipeline with the existing human body pose detection framework MediaPipe, followed by a keypoint classification algorithm. The action logic behind the Non-Player Character (NPC) is achieved with reinforcement learning, specifically Q-learning. We are presenting in this paper the related research, the modifications to the algorithms and our experiments on the new environment that we create.**

***Index Terms*—Gesture Detection, OpenCV, Openpose, MediaPipe, Reinforcement Learning**

Gesture Detection

I. Introduction

As video games developed, the ways of interaction are determined, such as mouse and keyboard. We would like to explore more ways to interact with games through machine learning. Hand recognition is an interesting and practical problem in the field of computer vision, as hands play a vital role in our interaction with the world. The technology to approach the task has evolved, from the sophisticated heuristics in traditional methods, to the recent powerful deep learning neural networks.

In recent years the field of object detection has made amazing progress, and the detection accuracy has advanced to approach or even exceed the human level. As a subtask of object detection, hand recognition also shared such improvement. The application of gesture detection also has been adopted in game designs. From simple website games, to mobile games, to virtual reality games, gesture detection has brought the unique and holistic experience of games to a completely new level.

Therefore, we explored this advancement of gesture detection technology, and utilized it in our game project. Specifically we experimented with three different frameworks capable of doing hand detention, namely OpenCV, Openpose, and MediaPipe. We compared their features, runtime performance and accuracy, and decided to adopt MediaPipe's hand keypoint detector in our architecture. Then we made gesture classification based on the keypoint information from the MediaPipe detector. Players of our game are able to use gestures to control their characters to complete tasks in the game, in real-time.

Certainly, there are people trying to bring the players more immersive experiences, like AR and VR, however, a lot of them rely on additional devices such as headsets and controllers. We think it may require extra money and time for players to buy and learn how to use the device. Thus, we want to free the players' hand and enable them to control the character only with their gestures.

Although gesture detection could be a lot more interesting to experience with hardware such as Kinect and depth camera, considering the cost of depth camera and the computing power of most computers, we decided to utilize 2D gesture detection algorithms on RGB images for our game. Therefore, we are aiming to provide players with an intuitive and immersive experience without requiring additional devices and skills. We would like the gesture detection algorithm to be able to run on CPUs and real time detection results could be transferred to the Unity for calculations and operations in the game.

II. Related Works

A. Traditional method using OpenCV

Before deep learning became the jewel in the crown of computer vision, the traditional approach for the hand recognition task required complex calculations from strong priors and physical restrictions to infer which shapes are actually hands. The resulting algorithm is still likely to be not adaptive enough to various natural conditions like the light condition. In addition, depth sensors were usually used for the detection, as they provide additional information than ordinary cameras.

The traditional gesture detection method using OpenCV requires four steps: background subtraction, motion detection, contour extraction and gesture detection [1]. First we need to separate foreground and background. Here the background subtraction method uses the concept of running averages. We look at the particular scene for 30 frames and compute the running average over the current frame and the previous frames. The static part of the frame that is not moving is the background. Then we will subtract the background from the current frame to capture the foreground object.

Fig. 1: OpenCV Gesture Detection Fist and Palm

After obtaining the foreground object, we will use a threshold to detect the hand region. In this way, all the parts in one frame that belong to the background or parts that are not hand region will be painted as black. Only the hand region will remain. This step is called thresholding and motion detection.

The third step is contour extraction. After processing the frame and marking all the unnecessary parts in the frame as black, we will get the hand region in the frame. We want to draw a contour around the hand. When we get the contour of the hand, we will be able to move to the next step to make calculations on gesture detection.

The final step of this method is gesture detection. Here we will find the extreme points on convex hull covering the the contour of the hand that gives us the furthest euclidean distance. Then we will find the center of the palm from these furthest points and plot a circle using the center of palm with a radius equal to the distance from the center of palm to the furthest point on the contour. Then we will calculate the number intersection points on the circle that are also on the contour of the hand. Then we will be able to find the number of fingers on the gesture and then determine what the gesture is.

III. Data and Environments

The data we use are the RGB images in JPG format. They are people's selfie pictures taken by the front camera of a mobile phone, and one or two hands may or may not be present in each picture. Some illustrations are shown in Figure 3. These data demonstrate the targeting real world scenarios of our project, which helped us make design decisions and improve the architecture as a whole.

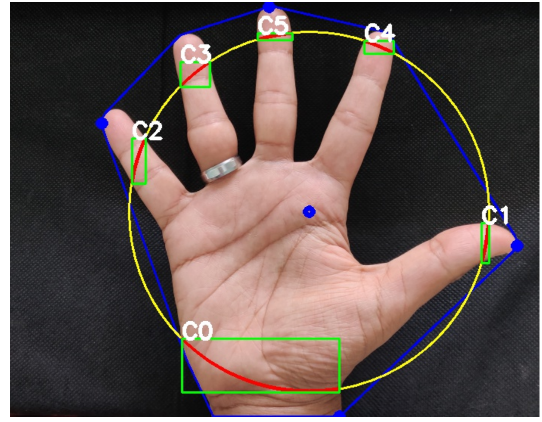


Fig. 2: Contour Box for Gesture Detection [1]

We used the Unity engine [2] to build the game logic and interfaces. The gesture detection algorithm described in this paper is the part that reads in images, and then tries to detect hand gestures in them. Then the game logic will determine what to proceed depending on the detection results.

IV. Methods - Openpose

A. Introduction

Openpose is a human body pose detector framework, proposed in 2018 by the Perceptual Computing Lab of Carnegie Mellon University [3]. It can jointly detect keypoints of human bodies, face, hands, and feet, with up to 125 keypoints for each person, on single images. It is capable of detecting multiple persons, and doing it in realtime (about 22 FPS) with a dedicated GPU card like the Nvidia GTX 1080 Ti.

B. Mechanism

Openpose uses a 21-keypoint model for hands, with four points for each finger plus one for the wrist. (show a figure) It performs the hand keypoint detection using an architecture called the Convolutional Pose Machine (CPM). CPMs were presented in a 2016 paper [4] by the Robotic Institute at CMU.

CPMs are neural networks designed for detecting structured objects. Structured objects refer to those objects that are composition of different parts with underlying arrangement between them. In the case of the hand detection problem, hands are formulated as a composition of keypoints identifying palms, knuckles, finger joints, and prints. Human hands are nimble, as a hand is able to take many different poses. However, in all these gestures, the position of each part is somehow physically related to the other parts. There is strong connection between these keypoints of fingers, given there is no serious damage with the hand.

CPMs begin with a backbone network to extract features from input images. The backbone refers to the first layers of

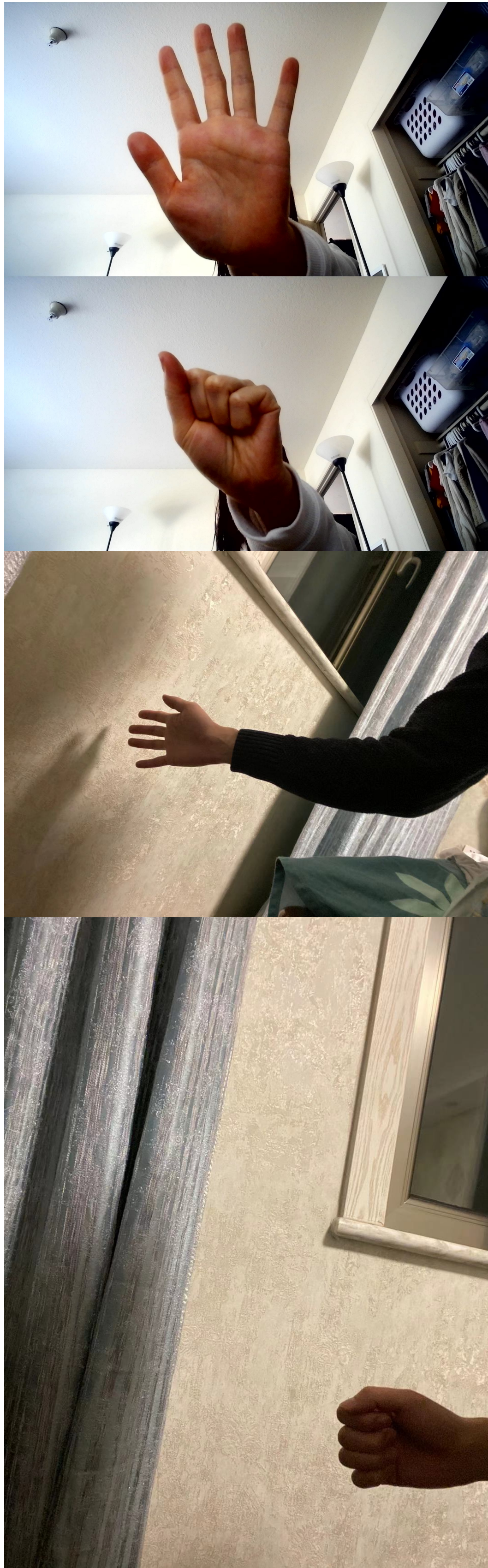


Fig. 3: Illustrations of Dataset Images

existing powerful networks trained on a huge dataset, such as AlexNet, VGG, and ResNet. They are able to extract features like shapes, texture etc. Then to infer the keypoints, CPMs consist of a sequence of stages, where each stage generates confidence maps, which are like heatmaps, for locations of all parts. Each stage takes as input both image features extracted by the backbone network and the confidence map created by its previous stage. Note that here the number of stages included in a CPM is not related to the number of keypoints in the problem. Confidence maps of all keypoints, one map for each keypoint, are created in each single stage. A larger or smaller number of stages will only affect the execution time and accuracy of the model. The more stages there are, the longer runtime and better accuracy is it expected to get.

CMU presented their hand keypoint detector in 2017, in this paper [5]. The backbone in their architecture is based on the first layers of pre-trained VGG-19 network [6] up to conv4_4, with two additional convolutions to produce a 128-channel feature map. This feature extraction is followed by six sequential prediction stages. The first stage takes the feature map and produces a set of 22 confidence maps, and one confidence map corresponds to each keypoint. The number of confidence maps is 22 instead of 21 as the number of keypoints, because one additional map is needed for identifying the background. It is shown the use of this additional confidence map helps the model improve accuracy by better distinguishing foreground objects and the background. Each stage after the first takes the output confidence maps of the previous stage concatenated by the feature map as input, and outputs a set of 22 keypoint confidence maps. The height and width of the feature map and confidence maps are designed to have a scale factor of 1/8 relative to the input images.

The loss function of each stage is a weighted L2 loss, which measures the distance between the predicted and ideal confidence map for each keypoint. It is designed to handle missing keypoint annotations by assigning zero weight to those cases in the function. Moreover, the total loss is accumulated through all six prediction stages, so that the training process is provided appropriate information about these intermediate layers in the considerably large network. Such design makes the network model better handle the problem of vanishing gradients. The problem of vanishing gradients refers to the observation of big networks, that the more there are intermediate layers in the network, the greater the magnitude of gradients decrease in the back propagation process.

C. Evaluation

As the first open-source realtime human pose detection system, Openpose with no doubt is an art with innovations and practicalities. However, after thorough exploration of its hand keypoint detector, we decided not to adopt it in our architecture. We present the following two reasons.

First, the standalone hand keypoint detector API in Openpose requires a region information as input, which should specify an area a hand would appear. This is not expected to be our use case, as we should allow our users to present their

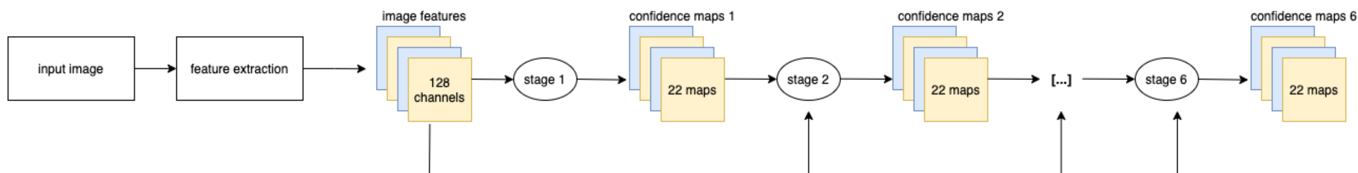


Fig. 4: Network Architecture of Openpose Hand Keypoint Detector

hands at any region in the camera scene. As an alternative, we also experimented with the full-functional human body pose keypoint detector. It does a great job detecting hands when other parts of the body like the arm and shoulder are in the scene with the hand. However, it is not able to detect floating hands where the arm or shoulder is not presented.

After investigation we discovered that it is because when performing hand detection, Openpose first makes region proposals based on position of identified wrists, arms, and shoulders. It uses the heuristic that hands are connected to these body parts. Then these proposed candidates are fed to the hand keypoint detector described above sequentially, where the correct detections will be generated. Such heuristic makes sense, but it still conflicts with our aim of a mobile application. Within the arm-length distance as people hold mobile phones in their hands, it is unreasonable to ask users to capture a whole arm just in order to get the hand detected with the front camera.

Second, the speed performance of Openpose can admittedly be near real-time, but that requires a machine with a powerful dedicated graphic card like the Nvidia GTX 1080 Ti. We tested the runtime of Openpose running on a moderate dedicated GPU as well as on CPU alone. It runs surprisingly slow on the laptop CPU. The average runtime that we tested for a single frame to process with Openpose is 4.3 seconds for CPU and around 43 milliseconds for GPU. As described before, our target platform is mobile phones, which generally are not equipped with powerful computing hardware like GPUs. Therefore, with the concern about runtime, Openpose should not be a good choice for us.

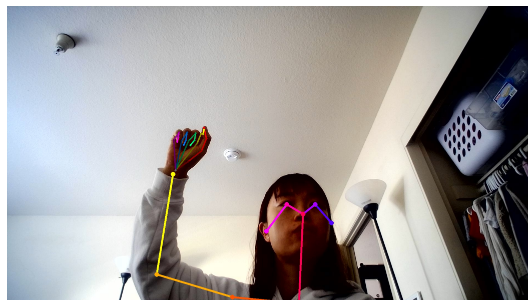


Fig. 5: Openpose Hand Detection Trial 1

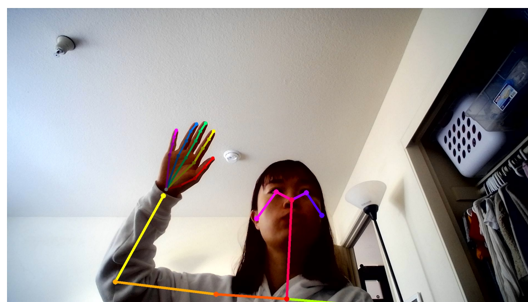


Fig. 6: Openpose Hand Detection Trial 2

V. Method - MediaPipe

A. Introduction

MediaPipe is a framework developed by Google to offer a cross-platform, customizable machine learning solution for live and streaming media [7]. It consists of several modules to serve as different tasks, such as the face detection, Iris detection, hands detection, pose detection, hair segmentation, motion detection, box tracking, object detection, etc. These modules can be used on Android, iOS, C++, Python, or Javascript.

MediaPipe can provide end-to-end fast machine learning inference to highly accelerate the processing speed even on common hardware; once built, the unified solution can be deployed anywhere; it offers ready-to-use cutting-edge solutions which demonstrates full power of framework; and the framework and solutions are both under Apache 2.0, free and open source.

B. Hand Module

In our project, we only use the MediaPipe Hands module to detect the hand gesture. The algorithm of MediaPipe is first proposed by Fan Zhang in Google Research [8]. This module offers the ability to perceive the shape and motion of hands, and users can deploy it as an important component to further their own project. For example, it can form the basis for sign language understanding and hand gesture control.

Normally, real-time hand detection is a challenging computer vision task because hands often occlude themselves or each other and lack high contrast patterns, but MediaPipe Hands is a high-fidelity hand and finger tracking solution. This module achieves real-time performance on a mobile phone, and even scales to multiple hands, while the other state-of-art approaches rely mainly on the powerful desktop platform.

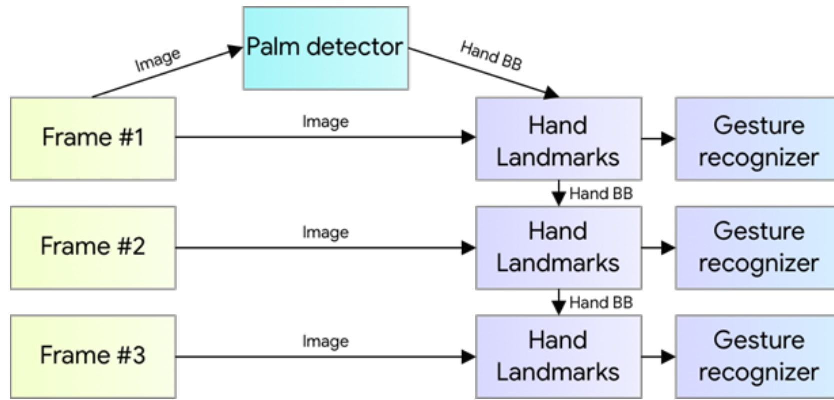


Fig. 7: MediaPipe Hands Pipeline

MediaPipe Hands machine learning pipeline consists of two models, which are a palm detection model that operates on the full image of hand gesture and returns an oriented hand bounding box, and a hand landmark model that operates on the cropped image region defined by the palm detector and returns high-fidelity 3D hand keypoints.

In the second step, the hand landmark model can receives the cropped hand image generated either from the palm detection model or based on the hand landmarks identified in the previous frame, which drastically reduces the need for data augmentation (e.g. rotation, translation and scale) and instead allows the network to dedicate most of its capacity towards coordinate prediction accuracy.

C. Palm Detection

The palm detection model works only on the detection of the initial hand locations. A single-shot detector model optimized for mobile real-time uses. The developer has trained a palm detector instead of a hand detector since estimating bounding boxes of rigid objects like palms and fists is significantly simpler than detecting hands with articulated fingers; and the non-maximum suppression algorithm works well even for two-hand self-occlusion cases as palms are smaller objects.

Moreover, the number of anchors can be reduced by a factor of 3-5 because the palm can be modelled using square bounding boxes, an encoder-decoder feature extractor is used for big scene context awareness even for small objects, and we minimize the focal loss during training to support a large amount of anchors resulting from the high scale variance. Overall, the palm detector can achieve an average precision of 95.7% in palm detection.

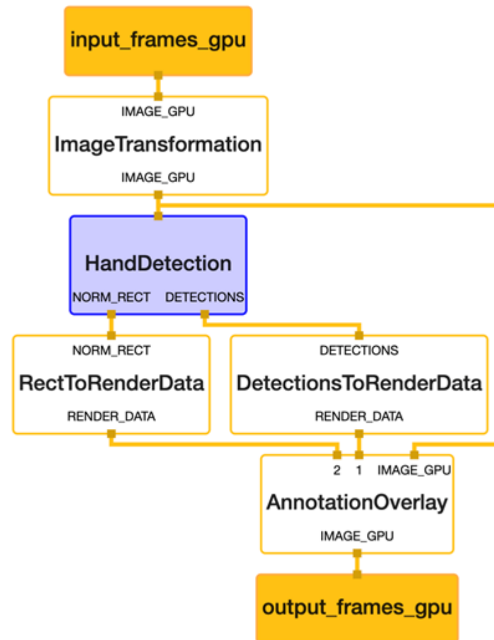


Fig. 8: MediaPipe Hands Palm Detection

D. Finger Detection

The second step, hand landmark model, performs precise keypoint localization of 21 3D hand-knuckle coordinates inside the detected hand regions via regression. The model learns a consistent internal hand pose representation and is robust even to partially visible hands and self-occlusions.

MediaPipe have manually annotated more than 30,000 real-world images with these 21 3D coordinates in order to obtain ground truth data, as shown below. To better cover the possible hand poses and provide additional supervision on the nature of hand geometry, MediaPipe also render a high-quality synthetic hand model over various backgrounds and map it to the corresponding 3D coordinates.

E. MediaPipe ML Pipeline

1) BlazePalm: A palm detector model (called BlazePalm) that operates on the full image and returns an oriented hand bounding box.

First, a palm detector is trained instead of a hand detector, since estimating bounding boxes of rigid objects like palms and fists is significantly simpler than detecting hands with articulated fingers. In addition, as palms are smaller objects, the non-maximum suppression algorithm works well even for two-hand self-occlusion cases, like handshakes.

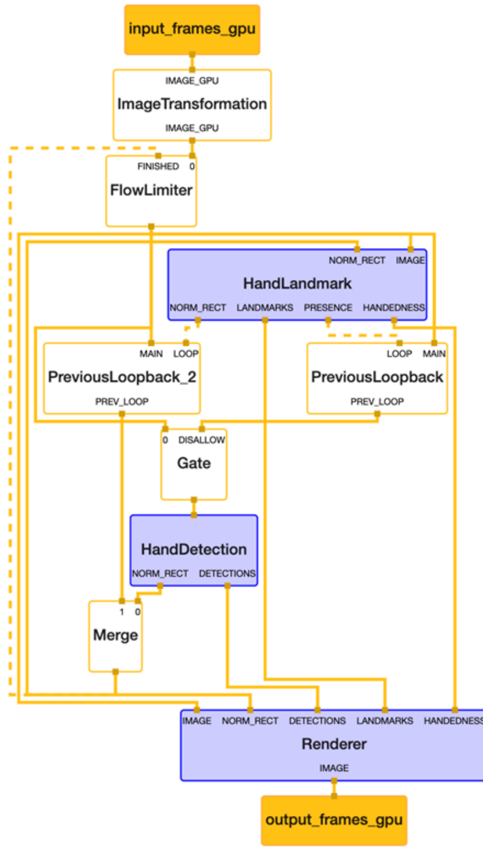


Fig. 9: MediaPipe Hands Fingers Detection

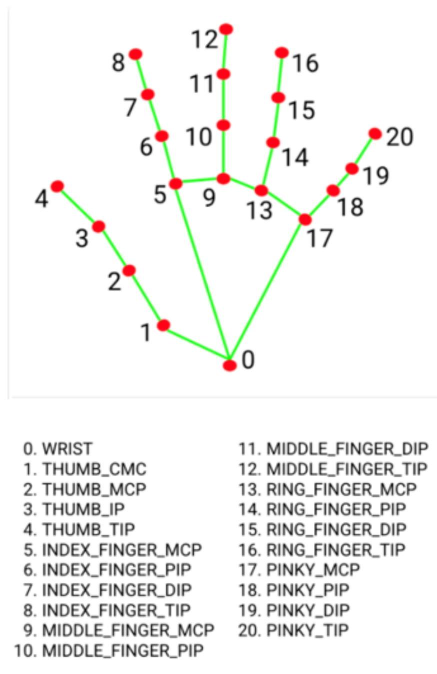


Fig. 10: Hand landmarks Location and Numbering in MediaPipe Hands

Moreover, palms can be modelled using square bounding boxes (anchors in ML terminology) ignoring other aspect ratios, and therefore reducing the number of anchors by a factor of 3-5. Second, an encoder-decoder feature extractor is used for bigger scene context awareness even for small objects. Lastly, the focal loss is minimized during training to support a large amount of anchors resulting from the high scale variance.

2) Hand Landmark Model: A hand landmark model that operates on the cropped image region defined by the palm detector and returns high fidelity 3D hand keypoints.

After the palm detection over the whole image our subsequent hand landmark model performs precise keypoint localization of 21 3D hand-knuckle coordinates inside the detected hand regions via regression, that is direct coordinate prediction.

3) Gesture Recognition: A gesture recognizer that classifies the previously computed keypoint configuration into a discrete set of gestures.

First, the state of each finger, e.g. bent or straight, is determined by the accumulated angles of joints. Then we map the set of finger states to a set of pre-defined gestures. This straightforward yet effective technique allows us to estimate basic static gestures with reasonable quality.

4) Implementation via MediaPipe : With MediaPipe, this perception pipeline can be built as a directed graph of modular components, called Calculators. Mediapipe comes with an extendable set of Calculators to solve tasks like model inference, media processing algorithms, and data transformations. Individual calculators like cropping, rendering and neural network computations can be performed exclusively on the GPU.

VI. Modifications to the Existing Environment

A. Gesture Classification with MediaPipe

Since Unity needs to take in an integer representing the type of gesture displayed by the player, we need to classify the output landmarks from MediaPipe into 'fist' (integer 0) and 'palm' (integer 1) in order to pass this integer to Unity. This classification method is based on the output results of MediaPipe gesture detection.

The output of MediaPipe has 21 points and each point has normalized x, y and z coordinates. These 21 points represent the key landmarks on one hand, from the bottom edge of the palm to the joints and end points of each finger. For classification of hand gestures in 2D we just need x and y coordinates to do the calculations.The output from MediaPipe is an object including the list of coordinates for each point. We need to process that output, extract the x and y coordinates for each point (landmark) and then put these information into an array.

For each finger on the hand, there is a state assigned to that finger. Based on the calculations of the relationships between the key joints on the finger, we will be able to tell if the finger is open or closed. For this calculation, we calculate the location of the joints on each finger and the top point of that

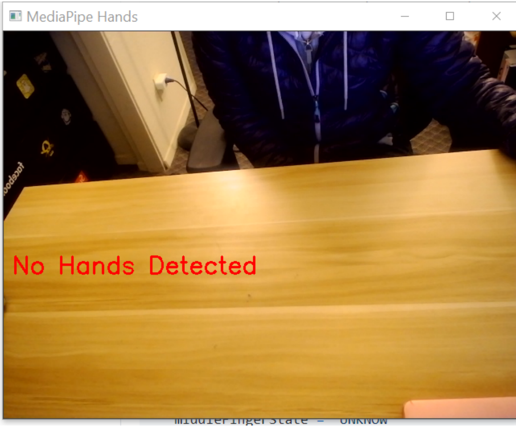


Fig. 11: MediaPipe Hands Testing: No Hands Detected

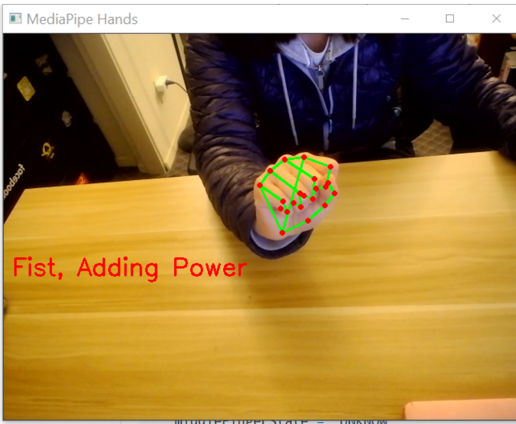


Fig. 12: MediaPipe Hands Testing: Fist

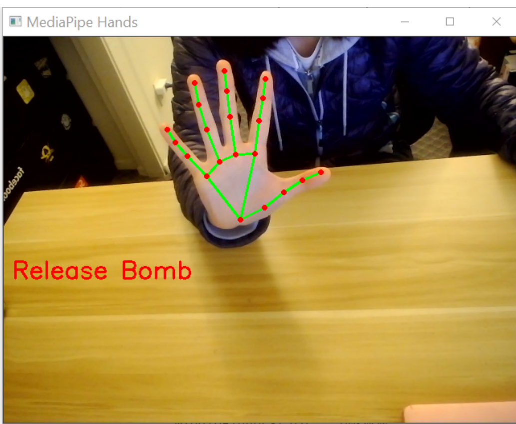


Fig. 13: MediaPipe Hands Testing: Palm

finger with respect to the connection point of that finger to the palm.

For the thumb, if the x coordinate of the tip point is smaller than the connection point for the left hand, or bigger for the right hand, then the thumb is in the closed position. Instead the thumb will be in an open position. For the four fingers, if we have the y coordinates of the tip point of that finger smaller than the connection point then that finger is in the closed position. Otherwise, the finger will be in an open position.

If all the fingers are closed, we will mark the gesture as 'fist'. This 'fist' gesture will be marked as integer 0 and will be passed to Unity to order the power gaining process before throwing the bomb. If one or more than one fingers are open, then we will mark the gesture as 'palm'. In this case, the 'palm' gesture will be marked as 1 and will be passed to Unity to order the bomb throwing process.

VII. Results and Analysis

Openpose does not work very well for our game where only the hand will show up in the camera and not the whole body of the person. We tested on the Openpose algorithm with only one hand showing in the frame and we didn't get output that satisfies the needs of our game. Also, Openpose runs very slow on the CPU. Although the Openpose model is well optimized on the GPU but consider that most computers do not have GPU installed so we decided to change to another gesture detection environment and we found MediaPipe.

MediaPipe works very well with computers just installed with CPU and for frames that just display single hand, MediaPipe could detect the hand in the frame efficiently. This satisfies our needs for gesture detection of our game design.

The whole gesture recognition algorithm is tested with our collected images as well as the live recording of a laptop front camera. Of the 100 testing images, the algorithm made correct classification on 92 of them. Figure 5 is some illustrations. It also runs smoothly on the live stream and is able to maintain 20 FPS. Reviewing the test results, we noticed that the size or scale of a hand in an image does affect detection accuracy, while its position in image does not. For example, when a person stands far away from the camera, his/her hand appears really small in the picture, then there is a chance the algorithm is not able to detect it. This should not be a problem for us since our use scenario is within an arm-length distance.

VIII. Limitations

There is still room for improvements in this study. First, this gesture classification process will not have stable detection results when the player puts his or her hand horizontally with four fingers pointing toward the x direction or with thumb pointing toward the y direction. Second, when the player places his or her hand up side down in the camera, this classification will not work properly.

In order to fix this problem, we need to identify the orientation of the hand based on the coordinates of the key landmarks on the palm and do a matrix multiplication to rotate and transform the coordinates of the 21 points on the hand.

This process will make the four fingers pointing upwards in the y direction. In this way, we will be able to make generalized classification on the gestures without considering multiple types of hand orientations.

IX. Conclusion and Future Work

Overall, we utilized the gesture detection techniques into our own game project. First, we tried to use OpenCV and Ppenpose for gesture detection, but then found that they cannot give good performance due to some limitations, so we decided to adopt MediaPipe. We use the MediaPipe Hands module in MediaPipe project to recognize two classification of gestures, which are fist or palm, and archives the accuracy of 92% given 100 testing images.

For the future work, we plan to refine our project to increase the accuracy of the gesture classification detection; meanwhile, we will try to build better connections between the gesture detection and unity.

Reinforcement Learning

I. Introduction

Reinforcement learning is a branch of machine learning to study how intelligent agents take action to maximize their rewards in a known or even unknown environment. It is widely used in game design because the problem it solves is exactly what agents in games do, taking actions to achieve some goal in a given environment. Among different approaches in reinforcement learning, Q-learning is a basic but effective algorithm to learn the value of an action in a particular state. It identifies an optimal policy given any finite Markov decision process, by computing the expected rewards for an action in a given state.

We chose to use Q-learning to implement the action logic behind the opponent agent in Bomb in Hand. The game has a finite and discrete map, and the agent has an explicit goal of surviving from player's bombs. It can be well formulated into a Q-learning model as follows.

II. Q-learning Model

In our game, the Non-Player Character (NPC) will move to avoid getting hit by the player's bombs and try to avoid falling off the cliff boundary or holes (caused by the player's bombs). To add more heuristic to the movement of NPC and avoid the NPC from staying at the optimal location without any movements, we add some random movements to the NPC while at the same time the NPC will follow the optimal path to avoid getting hit by the bomb from the player and avoid falling off the battlefield.

The Q-Learning we adopt uses the Bellman equation, which is shown in Figure 14. In this equation, the NPC will take the old Q value from the Q table, multiplied by a learning rate, and learn from the updated reward and also the optimum future reward with a discount factor. The learning rate α ranges from 0 to 1.0. The discount factor γ ranges from 0 to 1.0. We tried multiple learning rates and discount factors on the NPC. In order to minimize the chance of NPC from falling off the holes and upper cliff boundary, we chose a learning rate α of 1.0 so that the NPC will learn fully from the updated reward, especially when a battlefield location has been changed to a hole after reward updates. We also chose a discount rate γ of 0 so that the NPC will only follow the reward table to determine the optimal path. We also assign a random explore rate of 10% which means 10% of the time the NPC will move without following the optimal path, adding more heuristic to the game.

A. Q Table

The Q Table is a state action table. For our game, the states represent the location of the NPC in each round of the game and the actions represent the movements the NPC will possibly take for each round. There are 4 grids in each row and 5 grids in each column so there are 5x4=20 states available. For each state the NPC will have four actions to choose from. These actions are going UP, DOWN, LEFT and RIGHT. Therefore, the Q table will have 20 states and 4 actions for each state. Example of a Q table is shown in below.

B. Grid Indexing

In the Q-Learning algorithm, the grid indexing systems in Q table and in Unity are shown in Figure 16. The indexing for Q table is similar to the array indexing in a two-dimensional array. Q table needs to match each grid location to a state and will need to stretch the whole table into one list of states. Therefore in Q table the grids are indexed using strings with the first character representing the row number and the second character representing the column number. The grid indexing in Unity is further simplified to integers and each time when a grid on the battlefield is eliminated by a bomb, Unity will pass the grid index to the Q Learning algorithm directly for the reward table updates.

C. Reward Assignment and Updates

For each round of the game, the reward table will be updated based on the new situation on the battlefield and the NPC will run the Q learning algorithm to figure out the destination to go to and the optimal path to take. The Q learning algorithm helps the NPC figure out the optimal path based on information about NPC location, player location and hole location. The destination of the NPC for each round is the grid with the largest number of rewards in the whole grid.

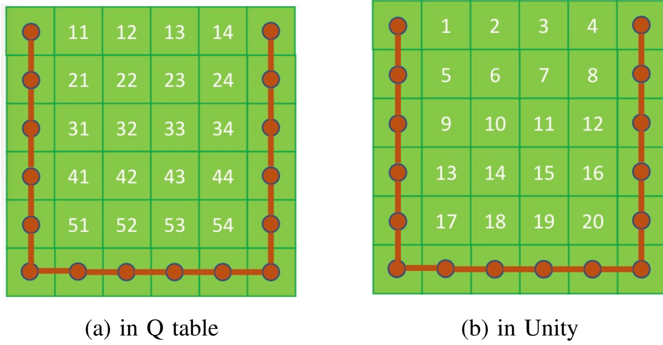
The reward table is updated based on the player location in the opposite battlefield, the distance of the NPC's location to the upper free boundary of the battlefield and the distance to the center of battlefield. Since the player will throw bombs in the up front direction to the NPC's battlefield, the NPC will need to avoid staying in the same column as the player. The further away the NPC stays from the same column with the player, the higher the rewards get and this reward is 100 points/step, meaning if the NPC stays 1 step away from the same column as the player, the reward will go up 100 points in the reward table. In addition, the NPC needs to stay away from the upper free boundary of the battlefield, and this reward is 40

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Fig. 14: Bellman Equation for Q-learning

	11	12	13	14	21	22	23	24	31	32	33	34	41	42	43	44	51	52	53	54
Up	-1000	-1000	-1000	-1000	-160	-1000	40	140	-140	-40	60	160	-120	-20	80	180	-60	40	140	100
Down	-140	-40	60	160	-120	-20	80	180	-60	40	140	240	0	100	200	300	100	100	200	100
Left	-160	-160	-1000	100	-140	-140	-40	60	-120	-120	-20	80	-60	100	100	140	100	100	100	100
Right	-1000	40	140	140	-40	60	160	160	-20	80	180	180	40	100	240	240	100	100	300	100

Fig. 15: Q Table Example



(a) in Q table (b) in Unity

Fig. 16: Grid Indexing Systems

points/step, meaning the NPC will get 40 points per step if the NPC stays away from the upper free boundary. Also, since it is more likely for the bomb to land in the middle of the battlefield compared to the upper and lower edges, NPC will get higher rewards by staying away from the center of the battlefield and this reward increment is 20 points per step. In order to prevent the NPC from staying at the optimum location with highest rewards, which is the lower left and lower right fence corner of the battlefield, we also added 50% heuristics to the reward updates so that there will be 50% of chances the NPC will explore the upper free boundary edge of the battlefield. The destination of the NPC will be the grid location in the reward table with highest reward.

For grid locations with holes, the reward will be -1000 and each round Unity will pass the updated hole location to the Q learning. If two grids are hit at the same time, both holes will be captured by Unity and then passed to Q learning algorithm for reward updates.

The upper free boundary will be assigned -1000 rewards. Since the battlefield is protected by fences on the left, right and lower boundary. We tried to adopt a strategy of the NPC taking the same rewards when it arrives at the grid location beside the fences and trying to take an action that goes towards the fence. For example, if the NPC arrives at the grid right above the lower fence and still tries to move downward, the NPC will stay in its original location and take the same reward at the original location. However, when testing this strategy we realize that the NPC will try to run into the fences and jitter a lot if the optimal strategy at that location is to take the local reward and then move towards the fences. Therefore, we changed the strategy to be -1000 rewards when the NPC arrives at the edge and still tries to move towards the fences (Figure 17), which means it is not recommended to the NPC to move towards the fences when it's close to the fences.

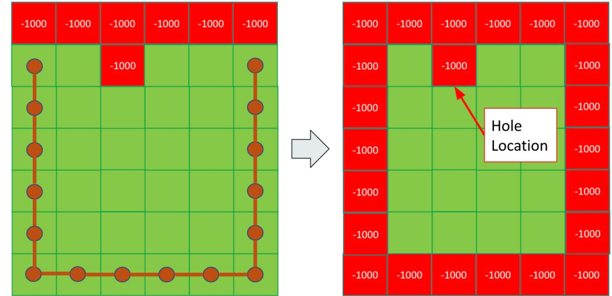


Fig. 17: Change Reward Strategy at Boundary Locations

D. The Optimal Route

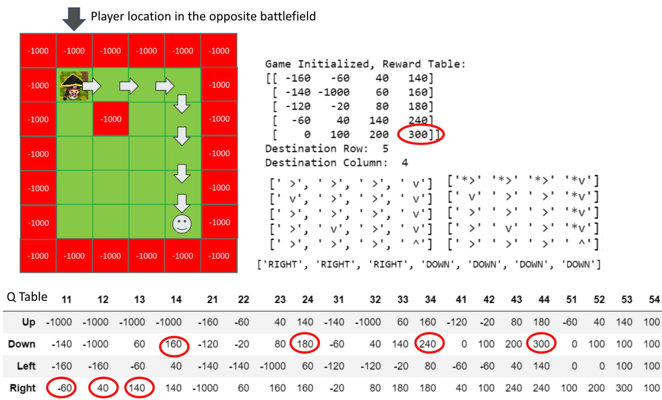
The Q learning algorithm will get player location, NPC location and hole locations from Unity during each round of the game. Reward table will be updated based on these parameters and the NPC will move towards the location with the maximum reward in the reward table. Then the Q table will be updated based on each action taken at each state. After updating the Q table, the action with maximum Q value in the Q table will be obtained. Then starting from the current location of the NPC, the optimal path of movement will be obtained by traversing the Q table with maximum Q values.

III. Results and Analysis

Two example runs of Q learning algorithm are shown in Figure 18 and 19. In the first example, the NPC is directed to move to the lower right corner of the battlefield where the maximum reward is located. And it determines the optimal route policy based on the computed Q table. In the second example, there is a hole right next to the NPC on its right, different from the first example, the NPC is directed to move downward first and then move to the lower right corner with max reward.

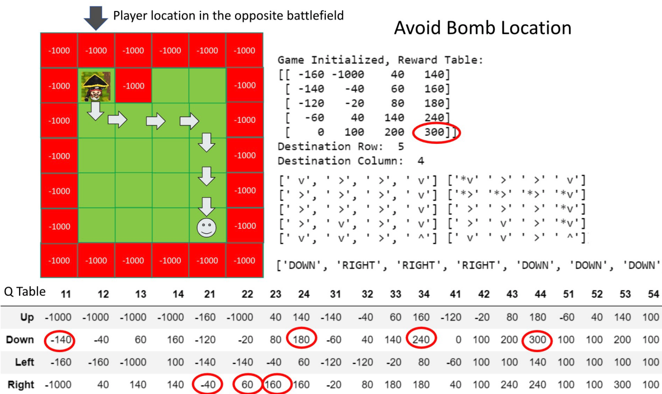
IV. Conclusion and Future Work

Overall, the agent we implemented using Q-learning does well on dodging the bombs from the player. Under the current game setting, the agent is able to find the path to get away if



Q Table	11	12	13	14	21	22	23	24	31	32	33	34	41	42	43	44	51	52	53	54
Up	-1000	-1000	-1000	-1000	-160	-60	40	140	-140	-1000	60	160	-120	-20	80	180	-60	40	140	100
Down	-140	-1000	60	160	-120	-20	80	180	-60	40	140	240	0	100	200	300	0	100	100	100
Left	-160	-160	-60	40	-140	-140	-1000	60	-120	-120	-20	80	-60	-60	40	140	0	0	100	100
Right	-60	40	140	140	-1000	60	160	160	-20	80	180	180	40	100	240	240	100	200	300	100

Fig. 18: Q-learning Example 1



Q Table	11	12	13	14	21	22	23	24	31	32	33	34	41	42	43	44	51	52	53	54
Up	-1000	-1000	-1000	-1000	-160	-1000	40	140	-140	-40	60	160	-120	-20	80	180	-60	40	140	100
Down	-140	-40	60	160	-120	-20	80	180	-60	40	140	240	0	100	200	300	100	100	200	100
Left	-160	-160	-1000	100	-140	-140	-40	60	-120	-120	-20	80	-60	100	100	140	100	100	100	100
Right	-1000	40	140	140	-40	60	160	160	-20	80	180	180	40	100	240	240	100	100	300	100

Fig. 19: Q-learning Example 2

such path exists. Meanwhile it also tries to avoid the holes on the ground. Thus, the winning strategy for the human player would be to play patiently to destroy the opponent's grid cell one after another, and trap the agent to a grid where it cannot move anymore. Then the player can win by bombing the standing target.

We would like to make the following improvements in the future. First, features like NPC throwing bombs should be added to make the game more amusing. Effort was put in but till now we have not found an effective way to make the agent dodge and attack at the same time, due to the natural contrariety of the two actions. Second, the player and opponent agent should be able to make free moves with various directions rather than the current grid moves. It requires a re-design of the Q-learning algorithm, as the size of Q-table will considerably expand. Finally, the reinforcement learning module is better incorporated into Unity. It is currently implemented in a separated Python module, which results in a communication delay. This avoidable delay should be eliminated for the seek of performance and user experience.

References

[1] G. Ilango, "Hand Gesture Recognition using Python and OpenCV," 2017. https://gogul.dev/software/hand-gesture-recognition-p1 (accessed Mar. 15, 2021).

[2] Unity, "Unity (software)." https://unity.com (accessed Mar. 15, 2021).

[3] Z. Cao, G. Hidalgo, T. Simon, S. E. Wei, and Y. Sheikh, "OpenPose: Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields," IEEE Trans. Pattern Anal. Mach. Intell., vol. 43, no. 1, pp. 172–186, 2021, doi: 10.1109/TPAMI.2019.2929257.

[4] S.-E. Wei, V. Ramakrishna, T. Kanade, and Y. Sheikh, "Convolutional Pose Machines," Jan. 2016, [Online]. Available: http://arxiv.org/abs/1602.00134.

[5] T. Simon, H. Joo, I. Matthews, and Y. Sheikh, "Hand Keypoint Detection in Single Images using Multiview Bootstrapping," Apr. 2017, [Online]. Available: http://arxiv.org/abs/1704.07809.

[6] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," Sep. 2014, [Online]. Available: http://arxiv.org/abs/1409.1556.

[7] Google, "MediaPipe" 2020. https://github.com/google/mediapipe (accessed Mar. 15, 2021).

[8] F. Zhang et al., "MediaPipe Hands: On-device Real-time Hand Tracking," Jun. 2020, [Online]. Available: http://arxiv.org/abs/2006.10214.

[9] Watkins, Christopher JCH, and Peter Dayan. "Q-learning." Machine learning 8.3-4 (1992): 279-292.